CompGC: Efficient Offline/Online Semi-honest Two-party Computation

Adam Groce
Reed College

Alex Ledger
Reed College

Alex J. Malozemoff
University of Maryland

Arkady Yerukhimovich
MIT Lincoln Laboratory

Abstract

We introduce a new technique, *component-based garbled circuits*, for increasing the efficiency of secure two-party computation in the offline/online semi-honest setting. We observe that real-world functions are generally constructed in a modular way, comprising many standard components such as arithmetic operations and other common tasks. Our technique allows circuits for these common tasks to be garbled and shared during an offline phase; once the function to compute is specified, these pre-shared components can be chained together to create a larger garbled circuit. We stress that we do *not* assume that the function is known during the offline phase — only that it uses some common, predictable components.

Improving on the above technique, we give a second method of chaining, which we call *single communication multiple connections* (SCMC) chaining, which allows blocks of consecutive wires holding multi-bit pieces of data to be connected between components with only a single transmitted wire label. This means that connecting components requires minimal communication.

Finally, we give an implementation, CompGC, of these techniques and measure the efficiency gains for various examples. We find that our techniques result in roughly an order of magnitude performance improvement over the best known standard garbled circuit-based secure two-party computation.

1 Introduction

Secure two-party computation allows a pair of parties, each with private input, to compute a function of those inputs without sharing them with each other. This is an extremely powerful tool, and it was shown by Yao to be feasible using an approach termed *garbled circuits* [30]. Since then, a long line of work has aimed to increase the efficiency of garbled circuit-based secure computation. This paper continues that effort.

In particular, our goal is to allow the use of offline pre-processing to significantly reduce online computation time for garbled circuit-based computation. This is not a new goal. Beaver, for example, showed how precomputation can significantly increase the online speed of the required oblivious transfers (OTs) [3]. Others have found similar ways to increase the online efficiency of the cut-and-choose technique needed for malicious security [13, 18, 19]. There is also a long history of precomputation in the setting of non-garbled circuit-based two-party computation [6, 25].

In the semi-honest setting in which all of our constructions work, it has long been known that precomputation can greatly increase efficiency *if the function is known ahead of time*, with only the inputs specified at the time of online computation. The protocol is simple: the garbler computes the entire garbled circuit ahead of time, with only OT computations (which can also be preprocessed, but still require some online communication), communication of the inputs, and evaluation done online. However, requiring that the function be known ahead of time is a substantial limitation.

In this work, we show a way to achieve a similar benefit *without* prior knowledge of what circuit will be computed. Towards this goal, we note that most functions of interest to compute securely are built in a modular way. Just as one would use functions in a programming language, the circuits for these functions use components that perform common tasks. There might be a portion of the circuit that takes the maximum of two numbers, for example, or that computes a hash function. We show that one can precompute garbled circuits for these smaller components and then chain them together in the online phase when the function to be computed is specified. We call this *component-based* garbled circuit construction. We show cryptographic protocols for carrying it out, and we provide an open-source implementation, CompGC, that achieves large efficiency gains, upwards of an order of magnitude improvement in online computation time, versus standard garbled circuit-based secure two-party computation.

We can imagine this system being used in two different ways. The first is to precompute garbled components of types that are common in a small class of functions from which the user knows the actual function will be drawn. For example, maybe the users know they will be computing edit distance between two genetic sequences, but they do not know what the length of each sequence will be. Even this small bit of uncertainty is sufficient to render the naive precomputation scheme discussed above useless. However, by sharing standard components that will be used for any such calculation, we make it so that the computation time once the inputs (including their lengths) are specified is greatly reduced. (Computing Levenshtein distance on arbitrary-length input is one of the use cases we investigate with experimental measurements.)

The second method is to develop an extensive library of components that would be useful for large classes of realistic functions. This would be useful if the parties in question are carrying out extensive computations over time, and are computing a wide array of possible functions. Here the library might include components for arithmetic operations, both simple (e.g., multiplication) and more complex (e.g., matrix inversion). It might contain common cryptographic primitives, text-processing methods, and other common subroutines, just as a standard library would for a programming language. And, just as with writing code, we expect most functions would need only minimal use of unstructured logic gates, instead relying mostly on these premade components.

Of course, these two methods are really just the two extreme ends of a spectrum. In between, there might be application-specific libraries that greatly increase the efficiency of cryptographic computations, or of machine learning computations, etc.

1.1 Our Contributions

We provide theoretical contributions describing two variants of component-based garbled circuits and showing that these constructions are secure. We then give a practical, open-source implementation, CompGC, that we use to experimentally show the significant efficiency improvements that these techniques can allow. Specifically, we make the following contributions.

Component-based garbled circuits. We give a protocol for precomputing garbled circuits for given components, and for combining these components as needed at runtime. We show that security is maintained by this protocol. This construction allows arbitrary linkage between component wires while requiring online communication of only *one* label per component input wire. We note that this technique is very similar to the "partial garbled circuits" of Mood et al. [22], although it was used for a different purpose in that work and, as described, required two labels per connection, whereas we only need a single label per connection.

Single communication multiple connections (SCMC) chaining. Our initial protocol specifies how each wire is connected between protocols and for each connected pair of wires sends a mask to the evaluator that is used to convert values in the appropriate way. However, frequently large sets of consecutive output wires represent single pieces of data (e.g., numbers or strings). These blocks of consecutive output wires are then mapped to an equal-length block of consecutive input wires in another component. We give a method for mapping consecutive output wires of one component to consecutive input wires of another while only sending a *single* value. We call this method "single communication, multiple connections" (SCMC) chaining, and it greatly reduces the communication cost of our protocol. This is somewhat analogous to the SIMD-style computation on multi-bit values that is done in the homomorphic encryption literature [29] and for GMW-based two-party computation [7, 28].

CompGC implementation. We develop our own standalone library `libgarble`[1] for garbling circuits. Our library is a based on the JustGarble implementation of Bellare et al. [4], but makes many internal improvements to the codebase. None of these improvements constitute theoretical improvements to the underlying algorithm, but rather optimizations of the code. For example, we revise the data structure by

[1]`https://github.com/amaloz/libgarble`

which circuits are stored in order to speed access to certain data. We believe this is a valuable contribution on its own, and is relevant even when not using our component-based precomputation strategy. Our library improves the performance of garbling and evaluating an AES circuit by 10% and 22%, respectively, as compared to JustGarble, along with many other improvements, including support for half-gates [31] and privacy-free garbled circuits [?] alongside a consistent API.

We then use `libgarble` as a building block to create a complete secure computation system, CompGC. This tool allows parties to precompute any specified library of components during the offline phase, using `libgarble` to garble each component. During the online phase, it creates a series of instructions for the evaluator that allows the chaining of the relevant components, and it handles the extra computation (outside of garbling and evaluation) that is required to distribute the input wire labels and decipher the output wire labels.

Experimental results. We use this implementation to conduct several experiments. We consider three settings: (1) computing AES using a single-round AES component as a building block; (2) using this single-round AES component to allow for encryption of arbitrary messages (of arbitrary length) using CBC mode; and (3) computing Levenshtein distance, which can be used for any number of applications, including text processing and genetic analysis. Here, again, we are eliminating the need to know the input length before computation. We measure total online time required to perform the secure computation over both localhost and a simulated realistic network configuration. In all of these measurements, we see substantial efficiency improvements due to precomputation. As an example, when computing Levenshtein distance between two 60 symbol strings, where each symbol comes from an 8-bit alphabet, we see a greater than order of magnitude improvement (from 10.6 seconds to 752 milliseconds to securely evaluate the function) when using our approach over the naive approach of sending the circuit online. See Section 7 for more details.

All of our work is done in the *semi-honest* model. We believe there are many use cases of secure computation for which semi-honest security is sufficient. For example, when two mutually trusting companies or agencies are prevented from sharing data by policy or legal restrictions, but otherwise trust each other to behave honestly. We also view semi-honest security as a natural stepping stone, and we expect these techniques can, with additional work, be extended to the malicious setting as well.

1.2 Paper Organization

The remainder of this paper is organized as follows. Section 2 summarizes the related prior work. Section 3 provides background information on garbled circuits and secure two-party computation, introducing the necessary notation that we use in the remainder of the paper. Section 4 describes our component-based garbled circuit technique. Then, Section 5 describes our improved single communication multiple connections garbled circuit chaining protocol. Section 6 provides the details on our prototype implementation of the described primitives and Section 7 gives the experimental results evaluating the performance of our schemes for several common classes of functions. We conclude in Section 8.

2 Related Work

Garbled circuits were first introduced by Yao in the 1980s [30] as a tool for general secure two-party computation. While they were originally viewed mainly of theoretical interest, this view has changed significantly over the past decade or so. Starting with the Fairplay system of Malkhi et al. [21], garbled circuits have been built into prototypes of secure computation. This has led to a long line of work (e.g. [4, 11, 12, 16, 19, 20, 22, 26]) that aims to improve the efficiency of garbled circuits and to build usable and practical systems for various real-world applications. Out of this work, the most efficient known implementations (not using specialized massively-parallel hardware [16]) of general garbled circuit-based computation are JustGarble [4] for security against a semi-honest adversary, and the "Blazing Fast 2PC" system [19] for the malicious setting.

One method for increasing the efficiency of garbled circuit-based secure computation is to work in the offline/online model and use preprocessing to reduce the online running time. A substantial line of work has focused on reducing the cost of the cut-and-choose technique [17] for malicious security using preprocessing [13, 18, 19]. However, all of these works require that the function to compute be defined *during* the pre-processing phase. Our goal is to allow the benefits of pre-processing *even when* one knows little about the function that might be computed.

In attempting to increase the online efficiency of secure computation, we are guided by many prior works that identified as a major bottleneck the time and bandwidth necessary to transmit the garbled circuit to the evaluator. Several works [14, 15, 24, 26, 31] aim to reduce the size of the circuit that must be communicated between the generator and evaluator. We see this paper as continuing this effort, aiming to reduce the amount of communication necessary in the online phase of garbled circuit evaluation. While we do not further reduce the overall size of the garbled circuit to be transmitted, we significantly reduce the amount of communication necessary in the online phase, after the function to compute and the inputs are chosen.

As communication is the main bottleneck, Gueron et al. [10] argue that the speed improvements made by JustGarble disappear due to the need to transmit the circuit. Because we send the circuit components in the *offline* phase, communication is no longer the bottleneck and we can thus reap all the performance benefits of using a JustGarble-based garbling library.

The idea of breaking circuits into smaller pieces appeared previously in the work of Mood et al. [22], where it was called "partial garbled circuits". Rather than use it to reduce online computation and communication time as we do here, Mood et al. used it as a way to reuse values in internal gates of a garbled circuit across multiple computations. Their technique also requires sending *two* correction labels per wire, whereas we can do it with just *one*. Also, our SIMD-style blockwise technique for chaining components together does not have a parallel in their work.

Finally, secure computation of single-instruction, multiple data (SIMD) operations has previously been studied in the context of homomorphic encryption [29] and in the context of secure computation based on the GMW protocol [7, 28]. However, to the best of our knowledge, our work is the first to apply this paradigm to garbled circuits. SCMC achieves significant savings in communication in a way that is very analogous to what is done with SIMD computation.

3 Preliminaries

In this section we briefly introduce the notation and key primitives that we use, as well as some background.

3.1 Garbled Circuits

Garbled circuits are the main tool used for all of our constructions. Our presentation here follows [10, 18] which is adapted from [5], and we refer the reader to those works for a more detailed presentation.

Garbled circuits, proposed originally by Yao [30], are a way of encoding a Boolean circuit that allow for secure evaluation of the function computed by that circuit. This encoding has the property that given encodings of values for each input wire, it is possible to evaluate the function computed by this circuit (i.e., learn the values of the output wires) without learning the values of the input wires or any of the internal circuit wires. This enables two-party secure computation where one party produces the garbled circuit and the input labels, and the other party evaluates the circuit to produce the output. This is described in more detail in Section 3.3.

More formally, a *garbling scheme* consists of two algorithms $(\textsc{Garble}, \textsc{Eval})$. On input a security parameter 1^κ and a circuit C, $\textsc{Garble}(1^\kappa, C)$ returns the triple (GC, e, d) where GC is the garbled circuit, e is the ordered set of input wire labels $\{(W_i^0, W_i^1)\}_{i \in \mathsf{Inputs}(C)}$, and d is the ordered set of output labels $\{(W_i^0, W_i^1)\}_{i \in \mathsf{Outputs}(C)}$.

Given a garbled circuit GC and a set of input labels $X = \{W_i^{x_i}\}_{i \in \mathsf{Inputs}(C)}$, $\textsc{Eval}(GC, X)$ computes the garbled output Z such that using the set d, it is possible to recover the actual output z (i.e., by finding Z in the ordered set of output labels).

Example. The most straightforward example of a garbled circuit is Yao's original scheme. Each wire w_i has two associated labels, W_i^0 and W_i^1, corresponding to values 0 and 1 respectively. For each gate there

w_0 label	w_1 label	w_{out} label	garbled table entry
W_0^0	W_1^0	W_{out}^0	$\mathsf{Enc}_{W_0^0}(\mathsf{Enc}_{W_1^0}(W_{out}^0))$
W_0^0	W_1^1	W_{out}^0	$\mathsf{Enc}_{W_0^0}(\mathsf{Enc}_{W_1^1}(W_{out}^0))$
W_0^1	W_1^0	W_{out}^0	$\mathsf{Enc}_{W_0^1}(\mathsf{Enc}_{W_1^0}(W_{out}^0))$
W_0^1	W_1^1	W_{out}^1	$\mathsf{Enc}_{W_0^1}(\mathsf{Enc}_{W_1^1}(W_{out}^1))$

Table 1: Garbled AND Gate. Only the values in the last column are sent to the evaluator. If the input wires have values a and b, then the evaluator knows W_0^a and W_1^b and can therefore decrypt $W_{out}^{a\wedge b}$.

is a table like Table 1. This table contains encryptions of the labels for the gate's output wire, using the labels of the input wires as keys. The encryptions are chosen so that the evaluator, knowing the labels of the two input wires, can decrypt the proper label of the output wire (and nothing else). Repeated evaluation of gates then propagates knowledge of the correct wire labels (for whatever initial input labels were given) through the entire circuit.

Privacy. In order to be useful for secure two-party computation, it is necessary that garbled circuits satisfy the following *privacy* notion. The values seen by the evaluator, GC, d, and X, should not reveal any information about x that is not revealed by the output $C(x)$. Formally, we require that there exist a polynomial time simulator S that on input $(1^\kappa, C, C(x))$ outputs a simulated garbled circuit that is indistinguishable from (GC, e, d) generated by Garble. Since S knows $C(x)$ but not x, this captures the fact that the output of Garble does not reveal anything (else) about x.

Free-XOR. Our constructions make use of one critical improvement to the original garbled circuits called free-XOR [15], which allows for XOR gates to be evaluated "for free" without requiring any garbled tables to be included in the garbled circuit. Specifically, this technique works by choosing a global random value R and then ensuring that the labels for all circuit wires have a difference of R. That is, for any wire w_i, $W_i^0 \oplus W_i^1 = R$. This enables the secure evaluation of an XOR gate by simply computing the XOR of the two incoming labels, as R cancels out appropriately.

3.2 Oblivious Transfer (OT)

Another key component for secure two-party computation is *oblivious transfer* (OT) [8, 27]. OT is a two-party primitive where one party (the sender) has as input two κ-bit strings (m_0, m_1) and the other party (the receiver) has a bit b. OT enables the receiver to receive m_b from the sender, while preventing the sender from learning which string was received (the value of b) and preventing the receiver from learning anything about m_{1-b}. In this paper we use the semi-honest OT construction by Naor and Pinkas [23].

One technique for optimizing OT that we make critical use of is OT preprocessing [3]. OT preprocessing allows splitting any OT protocol into an expensive offline phase and a much cheaper online phase. Specifically, in the offline phase, before the inputs are known, OT is performed on random inputs for both the sender and receiver. This requires a number of expensive cryptographic operations. However, in the online phase the pre-OT'd values are used to perform the OT on the parties' actual inputs without needing any additional expensive operations.

3.3 Secure Two-Party Computation

We now briefly describe how garbled circuits and oblivious transfer can be used to realize secure two-party computation. That is, to enable two parties to compute a joint function on their inputs without either party learning more than what is implied by its input and output. In this work we focus on two-party computation that is secure against a *semi-honest* adversary corrupting either of the two parties. That is, such an adversary follows the protocol as specified, but attempts to learn extra information from its interactions. For a formal treatment of the security of two-party computation we refer readers to the book by Goldreich [9].

In garbled circuit-based two-party computation of circuit C, we identify the two parties as the *garbler* who has input x and the *evaluator* who has input y. The garbler first runs $\textsc{Garble}(C)$ to produce (GC, e, d). He then sends GC and an encrypted form of d to the evaluator together with the wire labels corresponding to the bits of the garbler's input x. The encrypted form D of d corresponds to a random permutation of $\{\mathsf{Enc}_{W_i^0}(0), \mathsf{Enc}_{W_i^1}(1)\}_{i\in\mathsf{Outputs}(C)}$.

Now, for each bit of the evaluator's input y, the garbler and evaluator run an OT protocol by which the evaluator learns the appropriate wire label (without revealing that bit of y to the garbler). Now, the evaluator has all the inputs to run $\textsc{Eval}(GC, X)$ to recover the output wire labels. It then uses these wire labels to decrypt the entries in D to learn the appropriate output. If output by both parties is desired, the evaluator can send this output to the garbler.

4 Component-Based Garbled Circuits

As our first contribution, we introduce the concept of component-based garbled circuits to allow for much of the work involved in building and transmitting a garbled circuit to be done in an offline phase before the inputs or even the function to compute are known. This allows us to significantly improve the online performance of secure two-party computation schemes using garbled circuits. Our improvements stem from the observation that a common way to build circuits (and programs) is to compose them out of common building blocks or components. For example, common components such as circuits for arithmetic operations, cryptographic functions, and text processing can form the base for large classes of general computation.

We show how to take advantage of such common components for designing efficient garbled circuits. Specifically, our approach is to pre-garble a large number of common component circuits in an offline phase. Note that we do not need to know the computation to be performed (besides the generic components used to create said computation) or the inputs during this offline phase. Then, in an efficient online phase, we show how to *link* these components to form the actual circuit we wish to compute. The main benefit of this approach is that we only need to send a single wire label for each input wire of the components that the circuit is composed of. This is often much less than the size of the circuit to be computed. In fact, even if the components consist of single gates, our online communication corresponds to sending only *one* label per wire, which is half the size of the best known garbled circuit construction [31]. Since the time to communicate the garbled circuit is the major bottleneck, this leads to significant savings in the overall garbled circuit computation; see Section 7 for details.

More technically, a component-based garbling scheme is a triple of algorithms $(\textsc{Garble}, \textsc{Link}, \textsc{Eval})$. $\textsc{Garble}$ and $\textsc{Eval}$ are variants on the corresponding methods for standard garbled circuits, while $\textsc{Link}$ is new.

Garble. The $\textsc{Garble}$ procedure is unchanged, but now is given a component c as input (in place of a complete circuit C). $\textsc{Garble}(c)$ outputs the garbled component GC_c, input wire set e_c, and output wire set d_c, for this component.

Link. On input two garbled components $c_0 = (GC_0, e_0, d_0)$ and $c_1 = (GC_1, e_1, d_1)$ as well as a mapping of output wires of c_0 to input wires of c_1, $\textsc{Link}$ produces the *link* labels needed to convert from c_0 output wires to c_1 input wires. Specifically, suppose that output wire w_i of c_0 has labels (W_i^0, W_i^1) and input wire w_j of c_1 has labels $(\overline{W}_j^0, \overline{W}_j^1)$. Then, to link these two wires, $\textsc{Link}$ outputs $W_{ij} = W_i^0 \oplus \overline{W}_j^0$. Note that since we use the free-XOR optimization, we know that both $W_i^0 = W_i^1 \oplus R$ and $\overline{W}_j^0 = \overline{W}_j^1 \oplus R$ for some random value R. Therefore, we have that $W_i^0 \oplus \overline{W}_j^0 = W_i^1 \oplus \overline{W}_j^1$, so a single label W_{ij} is sufficient to connect both the zero and the one wire labels. This allows us to reduce the communication necessary to one label per component wire (together with a specification of which wire to link to which wire).

Eval. On input a list of garbled components $\{c_i\}$ and linking labels $\{W_{ij}\}$, $\textsc{Eval}$ computes the garbled outputs $\{Y_i\}$ as follows. Starting from the inputs, $\textsc{Eval}$ proceeds component by component, evaluating each component to get the component output wire labels. When appropriate, it uses these component output wire labels together with the appropriate link labels to recover the input labels for later components. Finally, once all the components are evaluated, EVAL recovers the garbled outputs $\{Y_i\}$ from the output components and uses d for that component to recover the (real) output y.

For details on the exact garbling scheme used to garble the components, the format for indicating which wires to link, and several further optimization improvements, we refer the reader to the implementation details in Section 6.

Privacy. We now show how to adapt the standard privacy definition for garbled circuits [5] to the component-based setting. Specifically, for a set of components $\{c_i\}_{i \in \mathsf{Components}}$, we want that the pre-garbled components $\{GC_i\}$, together with the input labels $\{W_j^{x_j}\}_{j \in \mathsf{Inputs}(C)}$, and the output map $d_{C_{out}}$ as well as all the link labels $\{W_{ij}\}_{i,j \in \mathsf{Components}}$ do not reveal any information about x. Formally, as in the case of garbled circuits, we require that there exist a polynomial time simulator $\mathcal{S}$ that on input $(1^\kappa, C, C(x))$, where $C(\cdot)$ is some polynomial size circuit, outputs simulated component garbled circuits for all components in C, input and output labels, as well as all the linking labels W_{ij} for linking all necessary wires that are indistinguishable from $(\{GC\}_i, e_{\mathsf{Input}(C)}, d_{\mathsf{Output}(C)})$ and W_{ij} generated by the real GARBLE and LINK procedures. Formally, security is captured by the following game:

The privacy experiment $\mathsf{Expt}^{\mathsf{priv}}_{\mathcal{A},\mathcal{S}}(\kappa)$:

1. Invoke adversary $\mathcal{A}$: compute $(C, x) \leftarrow \mathcal{A}(1^\kappa)$.
2. Choose a random $b \in_R \{0, 1\}$.
3. If $b = 0$: For each component c_i in C, compute $(GC_i, e_i, d_i) \leftarrow \text{GARBLE}(1^\kappa, c)$. Additionally, for each pair of components (c_i, c_j) that need to be linked, compute all the link labels $\{W_{ij}\} \leftarrow \text{LINK}(c_i, c_j)$. Finally, compute input labels $X = \{W_i^{x_i}\}_{i \in \mathsf{Inputs}(C)}$ and output map $d_{\mathsf{Output}(C)}$. Then output challenge $\tau = (\{GC_i\}, \{W_{ij}\}, X, d_{\mathsf{Output}(C)})$.
   If $b = 1$: Compute $\tau = (\{GC\}_i, \{W\}_{ij}, X, d_{\mathsf{Output}(C)}) \leftarrow \mathcal{S}(1^\kappa, C, C(x))$.
4. Give $\mathcal{A}$ the challenge τ and obtain a guess $b' \leftarrow \mathcal{A}(\tau)$.
5. Output 1 if and only if $b' = b$.

Definition 1. *A component-based garbled circuit scheme achieves* privacy *if for every probabilistic polynomial time A there exists a probabilistic polynomial time simulator S and a negligible function $\mu(\cdot)$ such that for every $\kappa \in \mathrm{N}$:*

$$\Pr\left[\mathsf{Expt}^{\mathsf{priv}}_{\mathcal{A},\mathcal{S}}(\kappa) = 1\right] \leq \frac{1}{2} + \mu(\kappa)$$

4.1 Component-Based Secure Two-Party Computation

We now briefly describe how to use component-based garbled circuits for secure two-party computation. In an offline stage, before inputs or even the computation to be performed are known, the garbler runs GARBLE on a number of components to pre-garble these components; it then sends $\{GC_i\}_{i \in \mathsf{Components}}$ and an encrypted form D of $d_{\mathsf{Output}(C)}$ (as specified in Section 3.3) to the evaluator. These components are circuit building blocks that comprise the eventual computation; however, their exact linking is not determined at this time. In parallel, the garbler and evaluator preprocess a number of instances of OT. Both the garbler and the evaluator store the received garbled components and preprocessed OTs.

When the function f to compute and the inputs (x, y) are known, the garbler assembles the circuit C out of the garbled components $\{c_i\}$. For each component pair that needs to be linked, the garbler runs $\text{LINK}(c_i, c_j)$ and sends the link labels W_{ij} along with the indices of the wires to be linked to the evaluator. Additionally, the garbler sends the input labels $\{W_i^{x_i}\}$ for the garbler's inputs. Finally, the garbler and evaluator complete the online phase of the OTs to retrieve the labels $\{W_i^{y_i}\}$ for the evaluator's input. Given this information, the evaluator runs EVAL to compute the circuit.

4.2 Analysis

To analyze the performance of component-based 2PC, we look separately at the online and offline phases. In the offline phase the garbling and transmission of garbled components is similar to the total communication normally done to garble and send a circuit. However, this communication can be done offline thus not effecting the online running time. The online phase, on the other hand, only sends one link label per pair of wires connecting any components. So, in total, the online communication necessary is just one label for each component input wire (along with information on which input wires map to which output wires). We note that, even in the case when components are just single gates, this still enables us to achieve communication of one label per gate (and XOR gates remain free). This is 50% savings over the best known construction [31] (again, discounting the metadata required to link these wires together). In the more realistic case, where components are substantially larger, the savings can be much greater.

4.3 Security

We now sketch a proof of security for our offline/online construction. Roughly, what we need to prove is that the added linking labels do not break the security of the original garbled circuit construction. More formally, we need to show a simulator that, given the output y, is able to generate simulated garbled components and linking labels that would look indistinguishable from the true garbled circuit.

We must consider the view of each party, where the "view" includes any messages received during the protocol. (Values computed and sent by a party themselves cannot give them additional information.) First we note that the view of the garbler in this construction only consists of its side of the OT protocol executions. This is the same as its view in the standard garbled circuit protocol, so no additional security argument is needed.

Next we consider security against a semi-honest evaluator. Roughly, we can use a slightly modified version of the standard garbled circuit simulator. This simulator produces a garbled circuit GC for the overall circuit C. The simulator then divides this circuit into components matching the components that were pre-garbled by the protocol. These garbled components are then modified as follows. For each output wire w_i of each linked component, a random label $\widehat{W_i}$ is chosen and is XORed with the output wire label. The result is a new label for each output wire. (The tables in the final gate before each output wire are modified to match the new values.) The output wires still have truly random labels, so these simulated values are still indistinguishable from the evaluator's true view. We now simply note that the random values $\widehat{W_i}$ for each component output wire serve as the simulated linking value that would connect each component's output to the relevant input wires of the next component. They have the same mathematical relationship to the wire labels as the true linking values do. Therefore the simulator has produced a complete simulation of the evaluator's view, and security is achieved.

5 SCMC Garbled Circuits

Building on our construction for component-based garbled circuits we introduce a new technique called *single communication multiple connections* (SCMC) chaining that allows further optimization of the online communication for a specific subclass of computations. Specifically, we observe that frequently large blocks of consecutive output wires really represent a single piece of data (e.g., numbers or strings). Thus, these blocks of consecutive wires are likely to be mapped in order to another component where they are used in further computation. Our SCMC construction takes advantage of this to provide much more efficient linking for such blocks of wires. Specifically, we send only *one* link label for the entire block, instead of one label per wire in our original construction. We note that we can also represent general computation in this way by introducing permutation components that rotate the wires for linking, but this will likely result in too many different components that need to be pre-garbled to be practical.

We achieve this performance gain by modifying how components are garbled (with security holding in the random oracle model). Specifically, we choose the labels for the input and output wires of all components in a fixed, correlated way. This is done in a way that guarantees that the differences between linked labels will be the same for all the wires in a block. Thus, sending this single difference is sufficient. Specifically, we slightly modify our construction as follows.

Garble. During garbling, for every component choose two random base values A and B and a random value T. We assume that all parties have access to a random oracle H. Then, set the input wire label W_i^b corresponding to wire i and bit b to $A \oplus H\left(T \oplus (i \| b)\right)$. For every output wire label, set it to $B \oplus H(T \oplus (i \| b))$.

Link. To link the output of component c_0 to the input of component c_1, it is sufficient to send the single link label $B_{c_0} \oplus A_{c_1}$. This achieves the desired effect since, for any wire i and bit b:

$$(B_{c_0} \oplus H(T \oplus (i \| b))) \oplus (B_{c_0} \oplus A_{c_1}) = A_{c_1} \oplus H(T \oplus (i \| b))$$

Eval. Same as before.

5.1 Analysis

The SCMC approach significantly reduces the communication needed for out component-based secure computation. Specifically, we now only need online communication complexity of a *single* label per block of wires to connect. In the extreme case where such blocks form entire components, this requires only *one link label per component.*

5.2 Security

We briefly sketch the proof of security. We prove security when H is modeled as a random function. Since our garbling library `libgarble` implements garbling schemes secure in the random permutation model [4], this does not introduce any additional assumptions.

As before (cf. Section 4.3), the simulation for a semi-honest garbler is trivial. The simulation for a semi-honest evaluator is exactly the same as before, except that instead of choosing random label $\widehat{W}_i$ *per wire*, we choose $\widehat{W}$ *per component*, and XOR this label with all output wire labels of the component. Now, $\widehat{W}$ servers as the linking value.

The security argument here similar to as before. Namely, the output wire labels of each component still have truly random labels. However, in this setting the evaluator can query the random oracle H, and can distinguish if it can guess T. As this value is random, this happens with negligible probability, completing the proof.

6 Implementation

We have implemented all the theoretical ideas discussed above in CompGC[2], a new system for secure computation with preprocessing. Here we describe the implementation in detail, and in the next section we present performance numbers from our experimental results.

CompGC uses as its primary building block the `libgarble`[3] library, which is based on the JustGarble implementation of Bellare et al. [4]. `libgarble` does just what its name implies — it creates a garbled version of a specified circuit and evaluates that circuit given inputs. It is a tool, rather than a complete implementation of secure computation. It does not carry out the oblivious transfers (OTs) necessary to share input, or the networked interactions necessary to send the garbled circuit (or the information for the OT protocols, or the output) between parties.

As `libgarble` is based on JustGarble, we made several improvements to the code, including cleaning up the API, improving the structures for storing the garbled circuit, etc. With these modifications, we can now evaluate an AES circuit in around 17 cycles/gate, a computation that takes around 22 cycles/gate on the

[2]`https://github.com/aled1027/2pc`
[3]`https://github.com/amaloz/libgarble`

same hardware with the original JustGarble implementation, an improvement of around 22%. Note that, while implemented in `libgarble`, we do *not* use the half-gates approach of Zahur et al. [31], which reduces the size of each garbled gate to two labels at the cost of two calls to the hash function H during evaluation. We instead use a scheme which requires three labels be transferred but only *one* call to H during evaluation. As we are only concerned with the online time, the benefits of a smaller circuit are outweighed by the extra cost in evaluation.

We then use `libgarble` to build CompGC. CompGC has both an offline and an online phase. In the offline phase, CompGC is given a library of components and computes a specified number of each component. This library could be small and special-built for a certain class of functions, or it could be a huge library of many common computational steps, to allow faster online computation of most realistic functions.

In the offline phase, the garbler side of CompGC uses `libgarble` to generate and garble the component circuits. The garbler saves the garbled component circuits, each tagged with a unique ID, and input and output labels to disk. The garbler side also sends the garbled component circuits and their unique IDs to the evaluator side, which saves the received data to disk. The offline phases finishes by performing the offline portion of OT preprocessing as described by Beaver [3].

We specify the function that the garbler and evaluator compute in the online phase with a JSON file. The file specifies what types of components are needed for the computation, and how the components' input and output wires should be connected. (Another format could be used to gain a small efficiency improvement, but we value the fact that the JSON file is human-readable.)

The garbler receives this function and the garbler's input to the function at the beginning of the online phase. It then generates a set of instructions for the evaluator. The instructions specify particular pre-shared garbled circuits (by ID, rather than just by type). The instructions also specify an order for their evaluation and specify how to feed the outputs of one component into the inputs of others. (This requires both specifying what wires connect where and specifying the relevant mask for each pair of components that are being connected. This is done using the SCMC connection techniques described in the previous section.) Finally, the instructions include the necessary information to convert the output wire labels to bits, as well as the wire labels for the garbler's input. The garbler sends these instructions to the evaluator.

Next, the garbler and evaluator perform the online phase of preprocessed oblivious transfer, resulting in the evaluator having input labels corresponding to its input. The evaluator now has all of the information necessary to perform the computation. It evaluates each component using `libgarble` (in an order specified by the instructions), and computes the input labels for each component from either input labels or processing the output of a previous component. Finally, the evaluator computes the final output (and can then send it back to the garbler).

7 Experimental results

We compared CompGC with the traditional setting where the entire circuit is transferred online. We implemented a semi-honest protocol using `libgarble` in which the parties preprocess OTs in an offline stage, but the circuit garbling and transfer is done online. This is the closest setting to our work, as we assume that the parties do not know which circuit they would like to compute until the online stage.

Experimental setup. All experiments were run on an Intel® Core$^{\text{TM}}$ i5-4210H CPU, and were conducted over two network settings. The first involved running both parties on the default localhost configuration, which on our machine has a latency of 0.012 ms and bandwidth of 35.2 Gb/sec. For the second network setting, we used the built in Linux emulator `netem` to configure localhost to have a latency of 33 ms (the average latency in the United States [1]) and a bandwidth of 50 Mbits/sec (more than the average bandwidth of 31 Mbits/sec in the United States as of September 2014 [2]). We chose to use a simulated network due to the ease of controlling the latency and bandwidth as well as the ease of reproducibility. Our implementation also requires reading data from disk: on our experimental machine we measured the cached reads speed as 7.6 GB/sec and the buffered disk reads speed as 96 MB/sec.

We ran four experiments: AES, CBC mode, and Levenshtein distance using both 30 and 60 symbols. We discuss each experiment in turn.

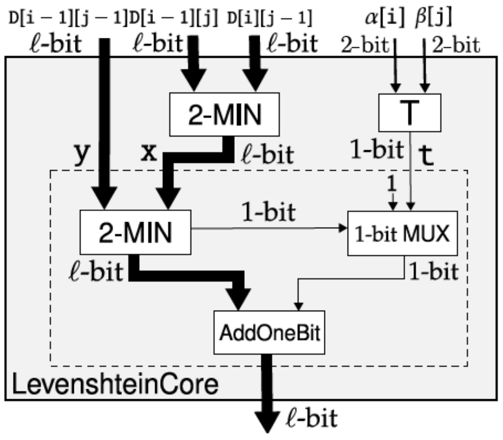

Figure 1: Levenshtein core circuit (taken from Figure 5(c) from the work of Huang et al. [12]).

	Time (localhost)		**Time (simulated network)**		**Total communication**	
	Naive	CompGC	Naive	CompGC	Naive	CompGC
AES	2.8 ± 0.03 ms	2.6 ± 0.2 ms	545 ± 0.5 ms	68 ± 0.1 ms	24 Mb	567 Kb
CBC mode AES, 10 blocks	36.8 ± 6.8 ms	21.5 ± 1.3 ms	4.8 ± 0.005 s	219 ± 1.2 ms	237 Mb	2.6 Mb
Levenshtein, 30 symbols	13.7 ± 0.2 ms	24.7 ± 1.0 ms	2.2 ± 0.0 s	318 ± 1.1 ms	108 Mb	6.3 Mb
Levenshtein, 60 symbols	69.6 ± 1.3 ms	67.0 ± 1.2 ms	10.6 ± 0.007 s	752 ± 1.8 ms	523 Mb	25 Mb

Table 2: Experimental results; see Section 7 for the experimental setup. Naive denotes our implementation of standard semi-honest 2PC using garbled circuits and preprocessed OTs using `libgarble`, whereas CompGC denotes our component-based implementation. Time is (online) computation time, not including the time to preprocess OTs, but including the time to load data from disk. All timings are of the *evaluator's* running time, and are the average of 100 runs, with the value after the ± denote the 95% confidence interval. The communication reported is the number of bits received by the evaluator.

AES: We treat each *round* of AES as a separate component. Thus, computing AES involves linking together 10 components (for each of the 10 rounds of AES when considering 128-bit inputs).

CBC mode: This algorithm provides a way of encrypting variable length messages using a blockcipher (in our case, AES) as an underlying building block. We use the same single-AES-round components as the above experiment, along with an XOR component. Our experiment involves running CBC mode over a 10 block message, and thus we use 110 components (100 for the AES rounds and 10 for the XOR components).

Levenshtein distance: This algorithm provides a measure of distance between two strings. We use as the core component the Levenshtein core circuit as explained by Huang et al. [12]; see also Figure 1. We use an 8-bit alphabet and run Levenshtein distance over strings containing both 30 and 60 symbols, which corresponds to 900 and 3600 components, respectively.

We note that these experiments are just a sample of what can be done using our tool. While the components we use are particular to our experiments, we note that, for example, an AES circuit could be used in other systems besides just CBC mode (e.g., any function that uses a blockcipher). Likewise, we could break the Levenshtein core circuit into its components (such as 2-MIN and AddOneBit; see Figure 1) which can likely be used in other circuit constructions.

Experimental results. Table 2 presents the results of the above experiments over both localhost and our simulated network. We compare the running times of both standard semi-honest secure two-party computation with the OTs preprocessed, which we denote as Naive, and our component-based garbled circuit protocol, which we denote as CompGC. We execute 100 runs of each experiment, reporting the average and

	Time (localhost)		**Time (simulated network)**	
	Naive	CompGC	Naive	CompGC
AES	2.8 ± 0.03 ms	1.1 ± 0.09 ms	545 ± 0.5 ms	67 ± 0.05 ms
CBC mode AES, 10 blocks	36.8 ± 6.8 ms	8.6 ± 0.5 ms	4.8 ± 0.005 s	204 ± 0.2 ms
Levenshtein, 30 symbols	13.7 ± 0.2 ms	14.8 ± 1.0 ms	2.2 ± 0.0 s	305 ± 0.2 ms
Levenshtein, 60 symbols	69.6 ± 1.3 ms	29.6 ± 0.6 ms	10.6 ± 0.007 s	709 ± 1.6 ms

Table 3: Experimental results without counting the evaluator time to load data from disk. See Table 2 for a description of the table.

	Time (simulated network)		**Total communication**	
	Standard	SCMC	Standard	SCMC
AES	135 ± 0.2 ms	68 ± 0.1 ms	656 Kb	567 Kb
CBC mode AES, 10 blocks	324 ± 1.1 ms	219 ± 1.2 ms	7.4 Mb	2.6 Mb
Levenshtein, 30 symbols	373 ± 1.0 ms	318 ± 1.1 ms	10.0 Mb	6.3 Mb
Levenshtein, 60 symbols	1131 ± 2.4 ms	752 ± 1.8 ms	44 Mb	25 Mb

Table 4: Comparison of our two approaches for communicating linked values: the standard approach (cf. Section 4) and the SCMC approach (cf. Section 5). The experiments are run on the simulated network as explained in Section 7; see Table 2 for description of the table.

the 95% confidence interval. We can see that when running on localhost, the timings are somewhat similar (Naive even beats CompGC for the case of Levenshtein distance using 30 symbols.) This is because in this setting communication is *not* the bottleneck, and thus the main benefit of our approach is not utilized. However, we can see the drastic improvement when considering a simulated network. Here we see an order of magnitude improvement of CompGC over Naive for CBC mode and Levenshtein using 60 symbols, as well as significant improvements for the other two cases. We can see why this is the case by looking at the total communication of each approach; CompGC demonstrates the greatest *time* improvement for those experiments with the greatest *communication* improvement.

Table 3 is similar to Table 2, except we remove the time to load the circuit components from disk from the evaluator's overall running time. This gives CompGC only a small improvement over CompGC in Table 2 on the simulated network, as the time to load data from disk is amortized away by the communication time. However, when running on localhost we see a significant improvement. This is because, as the communication time is minimal in this setting, the time it takes to load data from disk becomes the bottleneck.

Finally, Table 4 compares our first approach to chaining components discussed in Section 4 (termed the "standard" approach in the table) with the SCMC approach discussed in Section 5, again using the simulated network. We see an average of a 53% improvement of running time across all experiments when using SCMC over the standard approach. This is again due to the communication savings that result from only sending a single correction wire per *component* rather than one per output/input wire.

From these experiments, we confirm the belief that communication *is* the bottleneck for semi-honest secure two-party computation based on garbled circuits, and demonstrate that component-based garbling provides a powerful technique for reducing this bottleneck.

8 Conclusion

Our new technique, component-based garbled circuits, has greatly reduced online computation time for secure two-party computation. For functions we tested, the time needed for computation was reduced by an order of magnitude or more. This is done by drastically decreasing the amount of data that must be communicated during the online phase. While in principle one could construct functions for which our technique is unlikely to produce more than 50% savings with any realistic set of precomputed components, the benefit for *realistic* functions is much, much greater.

We have shown this in several cases where the general type of function is known ahead of time, but the specifics (e.g., input length) are not. However, the principle itself has much wider application than this. To make full use of our technique, libraries of circuits must be designed. These could be application-specific libraries for certain domains of computation, or there could be large, general-purpose libraries meant to provide useful components for most functions. Designing these sorts of libraries would also allow careful optimization of circuit size for each component.

We work only in the two-party and semi-honest settings, but multi-party and malicious settings could be amenable to a similar technique. We leave the task of designing specific protocols for these settings as future work.

Acknowledgments

Work of Arkady Yerukhimovich supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

Work of Alex J. Malozemoff conducted in part with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research.

References

[1] Global IP network latency. `http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html`. Accessed 2015-02-16.

[2] Measuring fixed broadband report – 2015. `https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-broadband-america-2015`. Accessed 2015-02-16.

[3] Beaver, D. Precomputing oblivious transfer. In *Advances in Cryptology – CRYPTO'95* (Santa Barbara, CA, USA, Aug. 27–31, 1995), D. Coppersmith, Ed., vol. 963 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 97–109.

[4] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy* (Berkeley, California, USA, May 19–22, 2013), IEEE Computer Society Press, pp. 478–492.

[5] Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. In *ACM CCS 12: 19th Conference on Computer and Communications Security* (Raleigh, NC, USA, Oct. 16–18, 2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM Press, pp. 784–796.

[6] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012* (Santa Barbara, CA, USA, Aug. 19–23, 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 643–662.

[7] Demmler, D., Schneider, T., and Zohner, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015* (San Diego, California, USA, Feb. 8–11, 2015), The Internet Society.

[8] Even, S., Goldreich, O., and Lempel, A. A randomized protocol for signing contracts. In *Advances in Cryptology – CRYPTO'82* (Santa Barbara, CA, USA, 1982), D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Plenum Press, New York, USA, pp. 205–210.

[9] Goldreich, O. *Foundations of Cryptography: Volume 2, Basic Applications*, vol. 2. Cambridge University Press, 2009.

[10] Gueron, S., Lindell, Y., Nof, A., and Pinkas, B. Fast garbling of circuits under standard assumptions. In *ACM CCS 15: 22nd Conference on Computer and Communications Security* (Denver, CO, USA, Oct. 12–16, 2015), I. Ray, N. Li, and C. Kruegel:, Eds., ACM Press, pp. 567–578.

[11] Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., and Wehrenberg, I. TASTY: tool for automating secure two-party computations. In *ACM CCS 10: 17th Conference on Computer and Communications Security* (Chicago, Illinois, USA, Oct. 4–8, 2010), E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM Press, pp. 451–462.

[12] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium* (San Francisco, California, USA, Aug. 8–12, 2011), D. Wagner, Ed., USENIX Association.

[13] Huang, Y., Katz, J., Kolesnikov, V., Kumaresan, R., and Malozemoff, A. J. Amortizing garbled circuits. In *Advances in Cryptology – CRYPTO 2014, Part II* (Santa Barbara, CA, USA, Aug. 17–21, 2014), J. A. Garay and R. Gennaro, Eds., vol. 8617 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 458–475.

[14] Kolesnikov, V., Mohassel, P., and Rosulek, M. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *Advances in Cryptology – CRYPTO 2014, Part II* (Santa Barbara, CA, USA, Aug. 17–21, 2014), J. A. Garay and R. Gennaro, Eds., vol. 8617 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 440–457.

[15] Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II* (Reykjavik, Iceland, July 7–11, 2008), L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 486–498.

[16] Kreuter, B., shelat, a., and Shen, C.-H. Towards billion-gate secure computation with malicious adversaries. In *21st USENIX Security Symposium* (Bellevue, Washington, USA, Aug. 8–10, 2012), T. Kohno, Ed., USENIX Association. Full version available at `https://eprint.iacr.org/2012/179`.

[17] Lindell, Y., and Pinkas, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007* (Barcelona, Spain, May 20–24, 2007), M. Naor, Ed., vol. 4515 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 52–78.

[18] Lindell, Y., and Riva, B. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology – CRYPTO 2014, Part II* (Santa Barbara, CA, USA, Aug. 17–21, 2014), J. A. Garay and R. Gennaro, Eds., vol. 8617 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 476–494.

[19] Lindell, Y., and Riva, B. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *ACM CCS 15: 22nd Conference on Computer and Communications Security* (Denver, CO, USA, Oct. 12–16, 2015), I. Ray, N. Li, and C. Kruegel:, Eds., ACM Press, pp. 579–590.

[20] Malka, L. VMCrypt: modular software architecture for scalable secure computation. In *ACM CCS 11: 18th Conference on Computer and Communications Security* (Chicago, Illinois, USA, Oct. 17–21, 2011), Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM Press, pp. 715–724.

[21] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay — a secure two-party computation system. In *13th USENIX Security Symposium* (San Diego, California, USA, Aug. 9–13, 2004), M. Blaze, Ed., USENIX Association.

[22] Mood, B., Gupta, D., Butler, K. R. B., and Feigenbaum, J. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *ACM CCS 14: 21st Conference on Computer and Communications Security* (Scottsdale, AZ, USA, Nov. 3–7, 2014), G.-J. Ahn, M. Yung, and N. Li, Eds., ACM Press, pp. 582–596.

[23] Naor, M., and Pinkas, B. Oblivious transfer with adaptive queries. In *Advances in Cryptology – CRYPTO'99* (Santa Barbara, CA, USA, Aug. 15–19, 1999), M. J. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 573–590.

[24] Naor, M., Pinkas, B., and Sumner, R. Privacy preserving auctions and mechanism design. In *EC* (1999), pp. 129–139.

[25] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO 2012* (Santa Barbara, CA, USA, Aug. 19–23, 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 681–700.

[26] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009* (Tokyo, Japan, Dec. 6–10, 2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 250–267.

[27] Rabin, M. O. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive 2005* (2005), 187.

[28] Schneider, T., and Zohner, M. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *FC 2013: 17th International Conference on Financial Cryptography and Data Security* (Okinawa, Japan, Apr. 1–5, 2013), A.-R. Sadeghi, Ed., vol. 7859 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 275–292.

[29] Smart, N., and Vercauteren, F. Fully homomorphic SIMD operations. Cryptology ePrint Archive, Report 2011/133, 2011. `http://eprint.iacr.org/2011/133`.

[30] Yao, A. C.-C. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science* (Toronto, Ontario, Canada, Oct. 27–29, 1986), IEEE Computer Society Press, pp. 162–167.

[31] Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology – EUROCRYPT 2015, Part II* (Sofia, Bulgaria, Apr. 26–30, 2015), E. Oswald and M. Fischlin, Eds., vol. 9057 of *Lecture Notes in Computer Science*, Springer, Heidelberg, Germany, pp. 220–250.